Refleksija u programskim jezicima

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Stefan Stanišić, Milan Ilić, Stefan Pantić, Dijana Zulfikarić

6. april 2015.

Sažetak

U ovom seminarskom radu obrađena je tema *refleksije* u programskim jezicima. Ukratko smo definisali sam pojam refleksije i osvrnuli se na istorijski aspekt koncepta od njegovog nastanka do upotrebe danas. Kroz primere jezika u kojima refleksija postoji pokazali smo njena osnovna svojstva i uobičajene načine primene. Kao poseban jezik izdvojili smo C++ i objasnili razloge zbog kojih refleksija nije ugrađena u samu konstrukciju programskog jezika, kao i alternativne koncepte pomoću kojih se ipak mogu postići neka od svojstava programskih jezika u kojima je refleksija omogućena.

Sadržaj

1 Uvod **2**
1.1 Podela refleksije . . . 2
1.2 Kada treba koristiti refleksiju i zašto? . . . 3

2 Istorija refleksije **3**

3 Podrška u programskim jezicima **4**
3.1 Java . . . 4
3.2 Python . . . 5
3.3 Prolog . . . 6
3.4 PHP . . . 7

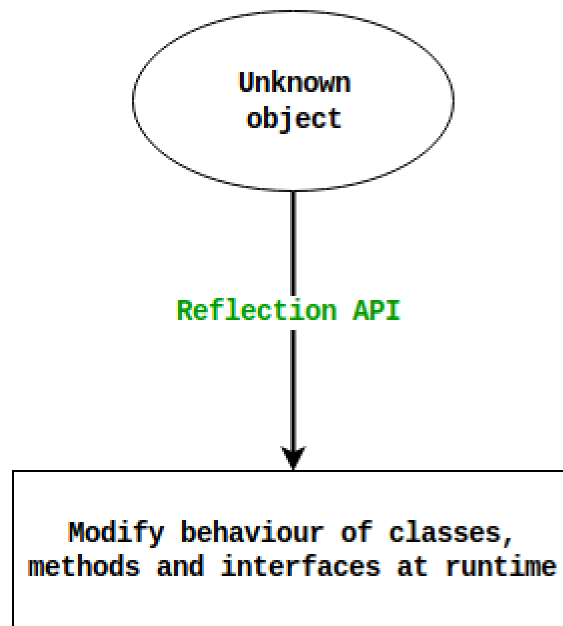
4 Refleksija u programskom jeziku C++ **8**
4.1 Programski jezik C++ . . . 8
4.1.1 Nastanak . . . 8
4.1.2 Osnovni koncepti C++-a . . . 8
4.1.3 Tipovi u programskom jeziku C++ . . . 8
4.2 Refleksija u C++-u . . . 9
4.2.1 Zašto C++ ne podržava refleksiju? . . . 9
4.3 RTTI (Run Time Type Identification) . . . 10
4.3.1 Operator typeid . . . 10
4.3.2 Primer RTTI-a . . . 10

5 Zaključak **11**

Literatura **12**

1 Uvod

Refleksija u oblasti programskih jezika se posmatra kao sposobnost računarskog programa da pregleda, ispita i modifikuje sopstvenu strukturu i ponašanje u toku izvršavanja. Na Slici 1 može se videti osnovna ideja refleksije kao koncepta.



Slika 1: Pojam refleksije

1.1 Podela refleksije

Postoje dve vrste refleksije: strukturna refleksija i refleksija ponašanja. Glavni razlog ovog razdvajanja jeste kompleksnost implementacije.

Strukturna refleksija se odnosi na sposobnost jezika da obezbedi kompletnu reifikaciju (proces koji pravi adresibilni objekat od onog koji to nije) kako programa koji se trenutno izvršava, tako i njegovih abstraktnih tipova. Strukturnu refleksiju je lakše implementirati od refleksije ponašanja, pa stoga programski jezici kao što su Lisp, Prolog, i Smalltalk godinama unazad sadrže mehanizme ove refleksije.

Refleksija ponašanja implicira sposobnost jezika da obezbedi kompletnu reifikaciju sopstvene semantike kao i podataka koji se koriste za izvršavanje programa. Još uvek nije dobro proučena, najviše zbog toga što se odnosi na upravljanje semantikom programa. Većina programskih jezika refleksiju ponašanja implementira preko tehnika interpretacije. Iz gore navedenih razloga refleksija ponašanja mnogo je manje rasprostranjena u programskim jezicima, međutim upravo je zbog toga glavni fokus naučnih istraživanja.

1.2 Kada treba koristiti refleksiju i zašto?

Refleksija se tradicionalno koristi za učitavanje modula ili klasa iz asemblera i kreiranje njihovih instanci u toku izvršavanja programa. Videti Primer 1. Klase predstavljaju prošireni koncept struktura podataka, pa kao i strukture podataka one mogu sadržati podatke, dok za razliku od struktura podataka mogu sadržati i funkcije. Predstavljaju jedan od osnovnih koncepata objektno-orijentisane paradigme.

Primer 1.1

```
1  // This will load the dll from a static resource
2  Assembly searchAssembly = LoadSearchServiceAssemblyFromResource();
3  // Now we need to find the class in the assembly
4  Type searchType = searchAssembly.GetType(searchAssembly.GetName().
       Name + "Search");
5
6  // If we have found the class
7  if (searchType != null) {
8      object searchResult = null;
9      dynamic classInstance = Activator.CreateInstance(searchType);
10     // DoSearch is a method declared on both SearchService Classes
11     var searchResult = instance.DoSearch();
12 }
```

Listing 1: Upotreba refleksije korišćenjem asemblera

Drugi razlog za korišćenje refleksije može biti za dobijanje javnih atributa objekta. Oni predstavljaju atribute kojima svi mogu da pristupe. Suprotni njima su privatni atributi kojima može da se pristupi samo u okviru te klase. Još jedan od načina upotrebe je za potrebe kreiranja privremenih objekata prilikom inicijalizacije u toku izvršavanja programa. Takođe, možemo je koristiti prilikom testiranja koda ili dela koda [8]. Uz to, svojstva refleksije mogu biti od značaja za kreiranje novih tipova u toku izvršavanja programa ili ispitivanje tipova u asembleru, što možemo videti u Primeru 2. Refleksija nam omogućava i promenu vrednosti polja koja su naznačena kao privatna u eksternim bibliotekama.

Primer 1.2

```
1  // Using Reflection to get information from an Assembly:
2      System.Reflection.Assembly integerTypeAssembly = typeof(System
       .Int32).Assembly;
3          System.Type integerType = typeof(System.Int32);
```

Listing 2: Korišćenje refleksije za ispitivanje tipova u asembleru

2 Istorija refleksije

Sam koncept refleksije proučavan je veoma dugo u filozofiji i realizovan do neke mere u logici. Prirodno je nastao u vestačkoj inteligenciji gde je tesno povezan sa krajnjim ciljem: refleksija je odgovorna za deo koji se smatra "inteligentnim ponašanjem". Iako su prvi računari programirani njihovim nativnim asemblerom koji je prirodno refleksivan, sam koncept refleksije u računarstvu i programiranju je uveo Brajan Kantvel Smit u svom doktorskom radu 1982. godine [11]. Kako bi formalizovao svoj koncept refleksije on je razvio dva programska jezika: 2-Lisp i 3-Lisp. Iako je on više pažnje posvetio relaciji reprezentacije koju je on zvao ϕ, njegov rad je ubrzo postao poznat u zajednici funkcionalnog programiranja zbog sposobnosti koju je 3-Lisp davao programima. Ta sposobnost se odnosila na to da programi mogu da "razmisle" o svojim izračunavanjima [1].

Jezik	**Java**	**C**	**Python**	**Prolog**	**Haskell**
Paradigma	OOP	Imperativna	Skript	Logička	Funkcionalna
Refleksija	✓	✗	✓	✓	✗

Tabela 1: Odnos programskih jezika, paradigmi i refleksije

Ovaj pristup inspirisao je veliki broj radova u zajednici funkcionalnog programiranja sve do kasnih 80-ih. Zahvaljujući njegivim konstruktima Lisp je postao poznat zbog svojih metalingvističkih sposobnosti. Od samog početka je omogućio manipulaciju i izvršavanje pojedinačnih delova programa koji su primitivna manifestacija koncepta refleksije. Lisp-ovi metacirkularni interpreteri su takodje bili predmet velikog broja istraživanja i debata. Stoga je Lisp zajednica imala veliko iskustvo što se tiče stvari koje je refleksija trebalo da omogući.

Do početka 90-ih postalo je jasno da je potrebno razviti strukturne mehanizme kako bi se u potpunosti savladala inherentna kompleksnost potpuno reflektivnog programskog jezika. Objektno-orijentisana paradigma se sama nametnula da preuzme taj izazov. Ceo jedan podskup zajednice objektno-orijentisanog programiranja bio je pod velikim uticajem Lisp-a, posebno deo zajednice vezan za Smalltalk i za objektno-orijentisanu verziju Lisp-a. Zbog toga ne iznenađuje činjenica da je je Smalltalk jedan od najreflektivnijih programskih jezika sa svojim "objekat na dnu"(eng. *object-to-the-bottom*) principom.

3 Podrška u programskim jezicima

Refleksija je podržana od strane mnogih programskih jezika i nije specifična za neku konkretnu programsku paradigmu, već se moze naći primer podrške u svakoj. U Tabeli 1 može se videti jedan od glavnih predstavnika svake programske paradigme i odgovor na pitanje da li je refleksija podržana u istom.

Najveći broj programskih jezika koji podržavaju refleksiju jesu oni koji su dinamički tipizirani, ali nisu retki ni ostali. U ovom poglavlju će biti više reči o podršci za refleksiju u sledećim programskim jezicima:

- Java 3.1
- Python 3.2
- Prolog 3.3
- PHP 3.4

3.1 Java

Programski jezik Java, objavljen 1996. godine od strane Sun Microsystems, je najkorišćeniji programski jezik objekno-orijentisane paradigme. Refleksija u Javi služi za ispitivanje i modifikaciju ponašanja metoda, klasa i interfejsa u toku izvršavanja programa. Potrebne klase za korišćenje refleksije se nalaze u *java.lang.reflect* [10] paketu i omogućavaju dobijanje informacija o klasi kojoj pripada prosleđeni objekat i metodama koje korisnik može da izvrši nad istim. Kroz refleksiju je takođe omogućeno pozivanje metoda klase bez obzira na njihovu vidljivost.

Refleksija omogućava dobijanje informacija o:

- Klasama - **getClass()** metoda vraća ime klase datog objekta

- Konstruktorima - **getConstructors()** metoda se koristi za dobijanje informacija o javnim konstruktorima klase objekta
- Metodama - **getMethods()** metod služi za dobijanje informacija o javnim metodama klase objekta

Primer 3.1

```
class Test
{
    public void method(int n)  {
        System.out.println("The number is " + n);
    }
}

public class Demo
{
    public static void main(String args[])
    {
        Test obj = new Test();

        Class cls = obj.getClass();
        System.out.println("Class name: " + cls.getName());
        // Class name: Test

        Constructor constructor = cls.getConstructor();
        System.out.println("Constructor name: " + constructor.
    getName());
        // Constructor name: Test

        Method[] methods = cls.getMethods();

        for (Method method:methods)
            System.out.println(method.getName());
        // method wait toString, ...

        Method methodcall = cls.getDeclaredMethod("method", int.
    class);

        methodcall.invoke(obj, 19);
        // The number is 19
    }
}
```

Listing 3: Primer korišćenja refleksije u Javi

3.2 Python

Programski jezik Python je dinamički tipiziran skript jezik koji je doživeo veliki porast popularnosti u prethodnih par godina i primenjuje se za širok spektar problema. Nastao je 1991. godine radom Gvida van Rosuma sa idejom čitljivosti koda.

Refleksija u Python-u je podržana kroz sledeće funkcije:

- **type(object, [bases, dict])** - služi za određivanje tipa promenljive ili kreiranje novog

Primer 3.2

```
x = 5
s = 'leskovac'
y = [1,2,3]
print(type(x)) # class 'int'
print(type(s)) # class 'str'
print(type(y)) # class 'list'
```

Listing 4: Određivanje tipa promenljive

- **isinstance(object, classinfo)** - proverava da li je objekat instanca ili podklasa date klase

Primer 3.3

```
class Foo:
    a = 5

bar = Foo()

print(isinstance(bar, Foo)) # True
print(isinstance(bar, (list, tuple))) # False
print(isinstance(bar, (list, tuple, Foo))) # True
```

Listing 5: Određivanje klase promenljive

- **callable(object)** - proverava da li objekat može biti pozvan (npr. ukoliko je funkcija). Klasa koja može biti pozvana mora imati definisan metod __*call()*__.

Primer 3.4

```
def baz():
  print('leskovac')

x = 5
y = baz

print(callable(x)) # False
print(callable(y)) # True
```

Listing 6: Provera da li je objekat funkcijski

- **dir(object)** - vraća listu atributa datog objekta

Primer 3.5

```
characters = ['a', 'b']
print(dir(number)) # 'append', 'reverse', ...
```

Listing 7: Određivanje atributa objekta

- **getattr(object, name)** - vraća vrednost datog atributa objekta

Primer 3.6

```
class Employee:
    salary = 25000
    company_name= 'lesko'

employee = Employee()
print(getattr(employee, 'salary')) # 25000
print(employee.salary) # 25000
```

Listing 8: Određivanje vrednosti atributa objekta

3.3 Prolog

Jedan od najznačajnijih predstavnika logičke paradigme jeste programski jezik Prolog, napravljen s namerom korišćenja u oblasti veštacke inteligencije. Postoje razne implementacije jezika od kojih su najpoznatije *B-Prolog*, *GNU Prolog*, *SWI-Prolog*, *Ciao*... Glavna korist refleksije u slučaju Prologa je dobijanje informacija o domenu promenljivih.

Sledeći predikati su samo neki koji primenjuju refleksiju u prologu:

- **fd_var(+Var)** - Tačno ako i samo ako promenljiva ima konačan domen
- **fd_inf(+Var, -Inf)** - *Inf* je infimum domena promenljive
- **fd_sup(+Var, -Sup)** - *Sup* je supremum domena promenljive
- **fd_size(+Var, -Size)** - *Size* je broj elemenata domena
- **fd_dom(+Var, -Dom)** - *Dom* je trenutni domen promenljive

3.4 PHP

PHP je skript jezik čija je originalna namena bila za upotrebu u veb programiranju. Za tvorca jezika se smatra Rasmus Lerdorf koji je 1995. godine objavio prvu verziju jezika. Danas je PHP osnova nekih od najposećenijih veb stranica, u koje spadaju i *www.google.com*, *www.facebook.com*, *www.yahoo.com*, *www.wikipedia.org...*

Jezik podržava reflektivno ponašanje i njega sprovodi kroz vrlo bogat reflektivni API [3] koji definiše sledeće klase:

- **ReflectionClass** - izveštava o informacijama vezanim za klasu
- **ReflectionFunction** - izveštava o informacijama vezanim za funkciju
- **ReflectionMethod** - izveštava o informacijama vezanim za metodu
- **ReflectionParameter** - izveštava o informacijama vezanim za parametre funkcije ili metode
- **ReflectionType** - izveštava o informacijama vezanim za tip

Primer 3.7

```
...
/** Profile */
class Profile {
    /** @return string */
    public function getUserName(): string
    {
        return 'Foo';
    }
}

class Child extends Profile {
}

$reflectionClass = new ReflectionClass('Profile');
$obj = new Profile();

var_dump($reflectionClass->getName()); // ['Profile']
var_dump($reflectionClass->getDocComment()); // ['/** Profile */']

$class = new ReflectionClass('Child');
var_dump($class->getParentClass()); // ['Profile']

$method = new ReflectionMethod('Profile', 'getUserName');
var_dump($method->getDocComment()); // ['/** @return string */']

var_dump($obj instanceof Profile); // bool(true)
...
```

Listing 9: Primer korišćenja refleksije u PHP-u

4 Refleksija u programskom jeziku C++

4.1 Programski jezik C++

C++ je multi-paradigmatski, statički tipiziran, kontekstno-slobodan programski jezik opšte namene. Podržava proceduralnu, objektno-orijentisanu, generičku i funkcionalnu paradigmu. Omogućava resursno nezahtevnu implementaciju apstrakcija.

Od svog nastanka 1980-ih godina do danas postao je jedan od najpopularnijih progamskih jezika, 4. mesto po 'Tiobe' [2] indeksu u vreme pisanja (Mart 2019.). Iako je prvobitno nastao kao jezik za sistemsko programiranje, zbog svoje ekspresivnosti, mogućnosti i odličnih performansi, našao je upotrebu u mnogim granama IT industrije uključujući razvoj video igara, mašinsko učenje, *end-user* aplikacije, serverske aplikacije i mnoge druge.

4.1.1 Nastanak

Kreator C++-a je Bjarne Stroustrup (*Bell Labs*) koji je imao ideju da proširi jezik C kako bi omogućio lakšu i širu primenu (prvobitni naziv jezika je bio "C sa klasama").

4.1.2 Osnovni koncepti C++-a

- C++ program se sastoji od niza tekstualnih fajlova (source i header fajlovi) koji u sebi sadrže deklaracije. Oni se prevode u cilju stvaranja izvršnog programa. Izvršavanje ovog programa pokrenuće *main* funkciju.
- Određene reči imaju specijalno značenje u kontekstu C++ programa i one se nazivaju ključnim rečima. Ostale reči mogu da se koriste kao identifikatori. Kompletan spisak svih ključnih reči može se naći na [4].
- Entiteti C++ programa su vrednosti, objekti, reference, funkcije, enumeratori, tipovi, članovi klasa, šabloni (templates), šablonske specijalizacije, prostori imena i paketi parametara (eng. *parameter pack*). Preprocesorski makroi nisu C++ entiteti. Entiteti se uvode deklaracijama koje im pripisuju imena i definišu njihove osobine. Deklaracije koje definišu sve osobine neophodne za upotrebu entiteta su definicije. Program mora da zadrži makar jednu definiciju i jednu ne-*inline* [5] funkciju ili promenljivu. Definicija funkcije sastoji se iz niza naredbi. Sva imena su vezana za svoju deklaraciju. Jedna od bitnih osobina koja se pripisuje deklaracijom je doseg [6].

4.1.3 Tipovi u programskom jeziku C++

C++ spada u grupu statički tipiziranih programskih jezika. Ovo znači da je tip svih promenljivih poznat prilikom kompilacije, kao i da se on ne može menjati prilikom izvršavanja. Jedna od glavnih prednosti ovog pristupa (u odnosu na dinamički tipizirane jezike) jeste to što kompajler može otkriti veliki broj bagova prilikom prevođenja.

Deklaracije u C++-u se sastoje od cv-kvalifikatora [7], specifikatora tipa, referenca i pokazvača.

Primer 4.1

```
const int &a; // Deklarise konstantnu referencu na promenljivu
    tipa int
```

Listing 10: Primer deklaracije promenljive

Svaka promenljiva, funkcija ili klasa mora biti eksplicitno deklarisana pre upotrebe. Ovom deklaracijom se njoj pripisuje tip (sa odgovarajućim kvalifikatorima). Ovaj tip je fiksan, u smislu da promenljiva jednom deklarisana kao promenljiva tipa A do kraja svog života ostaje tipa A. Detaljniji opis sistema tipova može se naći u zvaničnoj dokumentaciji.

4.2 Refleksija u C++-u

Postavlja se pitanje na kakav način je refleksija realizovana u C++-u? Odgovor je: ni na kakav. C++ kao programski jezik nema podršku za ono što potpada pod navedenu definiciju refleksije (1). Ovo ne znači da jezik nema nikakvu podršku sprovođenja upita o tipovima. Kao što je pomenuto, refleksija se odnosi na prikupljanje (i manipulaciju) tipova entiteta programa u toku njegovog izvršavanja. Kod C++-a ovo se odvija statički, tj. prilikom kompilacije samog programa. Mehanizmi jezika su takvi da se informacije o tipovima, manipulacije i dedukcije istih dešavaju prilikom prevođenja programa.

4.2.1 Zašto C++ ne podržava refleksiju?

Postoje brojni problemi kod dodavanja refleksije u C++:

- Zašto plaćati nešto što ne koristimo? Ovo se može navesti kao jedna od osnovnih filozofija dizajna C++-a. Zašto bi moj program nosio meta-podatke koje nikad neće biti iskorišćeni? Takođe, postojanje tih meta-podataka može da spreči kompajler u optimizaciji određenih delova koda. Nije vredno plaćati cenu postojanja meta-podataka zbog šanse da će možda nekada biti iskorišćeni.
- C++ ne daje nikakve garancije po pitanju kompajliranog koda, tj. dozvoljeno mu je da uradi šta god da želi dok god će rezultat imati očekivanu funkcionalnost. Ne postoji garancija da će deklarisana klasa stvarno postojati, kompajler može da odluči da je izbaci skroz ukoliko može da uvede semantički ekvivalentnu konstrukciju koja zahteva manje resursa. Fukcije mogu biti *inline*-ovane, a suvišne promenljive obrisane. U jeziku kao što je C# svaka deklarisana klasa, nezavisno od činjenice da li je ikada instancirana, će postojati u generisanom bajt kodu zajedno sa svojim meta-podacima. Kod C++-a ovo nije slučaj. Sve što može da se izbaci radi postizanja boljih performansi biće izbačeno. Brzina samog jezika zavisi od ovakvih agresivnih optimizacija.
- Šabloni u C++-u funkcionišu potpuno drugačije od *generics*-a u drugim jezicima. Svako instanciranje šablona pravi potpuno nov tip. *std::vector<int>* je potpuno različita klasa od *std::vector<double>*. Broj instanciranih tipova brzo naraste. Koju od ovih instanci refleksija treba da vidi? Da li *std::vector<int>*, *std::vector<double>*, ili sam šablon *std::vector*? To nije moguće zato što su šabloni *compile-time* konstrukcija [9]. Kada se uđe u sferu šablonskog meta-programiranja broj instanciranih tipova lako naraste na više stotina, s tim da bilo

koji od njih može biti uklonjen od strane kompajlera prilikom optimizacije. Šabloni se razrešuju prilikom kompilacije, tj. nemaju nikakvo značenje prilikom izvršavanja programa. Svaka od ovih klasa bi morala da bude vidljiva refleksiji, inače bi ona bila beskorisna. Postojanje meta-podataka za neinstancirane tipove narušava jednu od gore pomenutih filozofija samog jezika - zašto plaćati nešto što ne koristimo?

- Nije vitalna. Jezik kao C++ prosto nema neku preteranu potrebu za mehanizmom kao što je refleksija. Glavni razlog je šablonsko meta-programiranje. Iako ne može da reši svaki problem koji se rešava refleksijom, za većinu slučajeva u kojima bi se programer opredelio za korišćenje refleksije može se napisati meta-program koji postiže isti rezultat. Na primer, ukoliko su potrebne informacije o tipu *T* može se iskoristiti klasa *type_traits* iz biblioteke *boost*. U jezicima kao sto su C# i Java potrebno je "kopati"po klasi *T* da biste izvukli tražene informacije, što povlači određenu cenu izvršavanja.

4.3 RTTI (Run Time Type Identification)

Run Time Type Identification je možda nešto najbliže refleksiji što postoji u nekim implementacijama C++-a. Odnosi se na prikupljanje informacija o tipovima koje nisu mogle biti prikupljene prilikom kompilacije.

U C++-u pokazivači na klase imaju *statički* tip koji je pripisan prilikom deklaracije, kao i *dinamički* tip koji određen od strane tipa vrednosti na koju pokazuje. U Primeru 11 *statički* tip promeljive *foo* je A*, a *dinamički* je B*. RTTI omogućava programeru da dobije informaciju o *dinamičkom* tipu promenljive *foo*.

Primer 4.2

```
class A {};
class B: public A {};
extern B bar;
extern A* foo = &bar;
```

Listing 11: Primer RTTI

4.3.1 Operator typeid

Operator **typeid** vraća referencu na **typeinfo** objekat. Ovaj objekat opisuje *most-derived* tip objekta. Most derived tip objekta je tip najbliži tipu instanciranog. Glavna upotrebna vrednost ovog objekta je u poređenjima sa tipovima, paralela se može napraviti sa metodom *isinstance* programskog jezika Python. Još jedna mogućnost ovog objekta jeste utvrđivanje da li je neki drugi tip neposredno nasleđen od strane opisanog tipa. Iako RTTI postoji, njegova upotreba se izbegava i smatra se lošom praksom. C++ sadrzi veoma ekspresivan mehanizam za manipulaciju tipova (šabloni) koji treba koristiti umesto bilo kakvih *run-time* alternativa.

4.3.2 Primer RTTI-a

Deklarisaćemo dve klase od kojih će jedna nasleđivati drugu i iskoristiti RTTI da bismo dobili podatke o ovom odnosu u toku izvršavanja programa.

```
class Person {
  public:
      // ... Person members ...
      virtual ~Person() {}
};

class Employee : public Person {
      // ... Employee members ...
};
```

Listing 12: Deklarisanje polimorfne klase

Podaci o tipovima promenljivih person, employee i ptr su dostupni prilikom kompilacije i ne zavise ni od kakvog dinamičkog povezivanja.

```
Person person;
Employee employee;
Person *ptr{&employee};
// The string returned by typeid::name is implementation-defined
// Person (statically known at compile-time)
std::cout << typeid(person).name() << std::endl;
// Employee (statically known at compile-time)
std::cout << typeid(employee).name() << std::endl;
// Person * (statically known at compile-time)
std::cout << typeid(ptr).name() << std::endl;
```

Listing 13: Informacije poznate prilikom kompilacije

Međutim, tip polimorfne klase nije poznat prilikom kompilacije, ova informacija nastaje tek prilkom izvršavanja programa i stoga nam je potreban mehanizam RTTI da bismo je prikupili.

```
// Employee (looked up dynamically at run-time
//                  because it is the dereference of a
//                  pointer to a polymorphic class)
std::cout << typeid(*ptr).name() << std::endl;
```

Listing 14: Prikupljanje informacija o tipovima putem RTTI-a

Ovaj mehanizam može biti veoma koristan za prikupljanje informacija o hijerarhiji nedostupnih prilikom kompilacije, ali treba imati na umu utcaj na razumljivost i efikasnost koda (RTTI je veoma spor mehanizam), kao i na činjenicu da se upotreba generalno izbegava i smatra lošom prakсom (do te mere da mnogi savremeni kompajleri gase mogucnost bez eksplicitnog zahteva za njegovim omogućavanjem).

5 Zaključak

- U ovom radu predstavljen je pojam refleksije u programskim jezicima. Objasnili smo koncept refleksije i njenu podelu na tipove, kao i razloge za istu. Ukratko smo se osvrnuli na njen istorijski razvoj i videli da refleksija korene vuče još od prvih asemblerskih jezika.
- Bavili smo se odnosom refleksije i programskih paradigmi kroz primere u različitim programskim jezicima. Izabrali smo Javu, Prolog i PHP kao predstavnike objektno-orijentisane, logičke i skript paradigme, kao i Python kao jezik opšte namene.
- Posebno smo se bavili programskim jezikom C++. Naveli smo razloge zbog kojih refleksija nije implementirana u samom jeziku, kao i neke od alternativa refleksiji koje možemo koristiti ukoliko su nam njena svojstva ipak potrebna.

- Refleksija kao koncept može biti korisna za različite svrhe, međutim mora se oprezno koristiti. Činjenica je da, pored toga što je njeno korišćenje resursno zahtevno, sama upotreba refleksije daje programeru moćan alat koji se veoma lako može upotrebiti na neispravan način. Zbog toga je refleksija svoje mesto našla u mnogim programskim jezicima, dok su se drugi opredelili za alternativne pristupe bez njene implementacije u programski jezik.

Literatura

[1] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. 1995.

[2] Tiobe Foundation. Tiobe Index, 2018. on-line at: https://www.tiobe.com/tiobe-index/.

[3] The PHP Group. PHP Documentation, 2019. on-line at: https://www.php.net/manual/en/book.reflection.php.

[4] ISO. CPP Reference, 2017. on-line at: https://en.cppreference.com/w/cpp/keyword.

[5] ISO. CPP Reference, 2017. on-line at: https://en.cppreference.com/w/cpp/language/inline.

[6] ISO. CPP Reference, 2017. on-line at: https://en.cppreference.com/w/cpp/language/scope.

[7] ISO. CPP Reference, 2017. on-line at: https://en.cppreference.com/w/cpp/language/cv.

[8] M.G. Limaye. *Software Testing: Principles, Techniques and Tools*. McGraw Hill Education, 2009.

[9] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[10] Oracle. Java Documentation, 2018. on-line at: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html.

[11] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1982.